1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 1993	3. REPORT TYPE AND DATES COVERED technical

4. TITLE AND SUBTITLE Backtracking Techniques for Hard Scheduling Problems	5. FUNDING NUMBERS F30602-88-C [illegible]01
6. AUTHOR(S) Norman Sadeh, Katia Sycara and Yalin Xiong	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The Robotics Institute Carnegie Mellon University Pittsburgh, PA 15213	8. PERFORMING ORGANIZATION REPORT NUMBER CMU-RI-TR-[illegible]8
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA, McDonnell Aircraft Co., Digital Equipment Corp.	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; Distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This paper studies a version of the job shop scheduling problem in which some operations have to be scheduled within non-relaxable time windows (i.e. earliest/latest possible start time windows). This problem is a well-known NP-complete Constraint S[illegible]action Problem (CSP). A popular method for solving this type of problems consists in using depth-first backtrack s[illegible]rch. Our earlier work focused on developing efficient consistency enforcing techniques and efficient variable/value order[illegible] heuristics to improve the efficiency of this sea[illegible]ch procedure.

In this paper, we combine these techniques with new look-back schemes that help the search procedure recover from so-called deadend search states (i.e. partial solutions that cannot be complete[illegible]ithout violating some constraints). More specifically, we successively describe three intelligent backtracking sc[illegible]s: (1) *Dynamic Consistency Enforcement* dynamically identifies critical subproblems and determines how far to [illegible]track by selectively enforcing higher levels of consistency among variables participating in these critical subproblen[illegible], (2) *Learning From Failure* dynamically modifies the order in which variables are instantiated based on earlier conflicts, and (3) *Heuristic Backjumping* gives up searching areas of the search space that appear too difficult. These schemes are shown to (1) further reduce the average complexity of the search procedure, (2) enable our system to efficiently solve problems that could not be solved otherwise due to excessive computational cost, and (3) be more effective at solving job

14. SUBJECT TERMS	15. NUMBER OF PAGES pp
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT unlimited	18. SECURITY CLASSIFICATION OF THIS PAGE unlimited	19. SECURITY CLASSIFICATION OF ABSTRACT unlimited	20. LIMITATION OF ABSTRACT unlimited

Backtracking Techniques for Hard Scheduling Problems

Norman Sadeh, Katia Sycara and Yalin Xiong

[illegible]

Carnegie Mellon University

The Robotics Institute

THE ROBOTICS INSTITUTE

Technical Report

12

Backtracking Techniques for Hard Scheduling Problems

Norman Sadeh, Katia Sycara and Yalin Xiong

CMU-RI-TR-93-08

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

March, 1992

Also appeared as Robotics Institute technical report CMU-RI-TR-92-06

This research was supported, in part, by the Defense Advanced Research Projects Agency under contract #F30602-88-C-0001, and in part by grants from McDonnell Aircraft Company and Digital Equipment Corporation.

Table of Contents

List of Figures

List of Tables

Abstract

This paper studies a version of the job shop scheduling problem in which some operations have to be scheduled within non-relaxable time windows (i.e. earliest/latest possible start time windows). This problem is a well-known NP-complete Constraint Satisfaction Problem (CSP). A popular method for solving this type of problems consists in using depth-first backtrack search. Our earlier work focused on developing efficient consistency enforcing techniques and efficient variable/value ordering heuristics to improve the efficiency of this search procedure. In this paper, we combine these techniques with new look-back schemes that help the search procedure recover from so-called deadend search states (i.e. partial solutions that cannot be completed without violating some constraints). More specifically, we successively describe three "intelligent" backtracking schemes: (1) *Dynamic Consistency Enforcement* dynamically identifies critical subproblems and determines how far to backtrack by selectively enforcing higher levels of consistency among variables participating in these critical subproblems, (2) *Learning From Failure* dynamically modifies the order in which variables are instantiated based on earlier conflicts, and (3) *Heuristic Backjumping* gives up searching areas of the search space that are deemed too difficult. These schemes are shown to (1) further reduce the average complexity of the search procedure, (2) enable our system to efficiently solve problems that could not be solved otherwise due to excessive computational cost, and (3) be more effective at solving job shop scheduling problems than other look-back schemes advocated in the literature.

1. Introduction

This paper is concerned with the design of recovery schemes for incremental scheduling approaches that sometimes require undoing earlier scheduling decisions in order to complete the construction of a feasible schedule.

Job shop scheduling deals with the allocation of resources over time to perform a collection of tasks. The job shop scheduling model studied in this paper further allows for operations that have to be scheduled within non-relaxable time windows (i.e. earliest possible start time/latest possible finish time windows). This problem is a well-known NP-complete Constraint Satisfaction Problem (CSP) [Garey 79]. Examples of such problems include factory scheduling problems, in which some operations have to be performed within one or several shifts, spacecraft mission scheduling problems, in which time windows are determined by astronomical events over which we have no control, factory rescheduling problems, in which a small set of operations need to be rescheduled without revising the schedule of other operations, etc.

A generic approach to solving CSPs relies on depth-first backtrack search [Walker 60, Golomb 65, Bitner 75]. Using this paradigm, scheduling problems are solved through the iterative selection of a variable (i.e. an operation) and the tentative assignment of a value (i.e. a reservation) to that variable. If in the process of constructing a solution, a partial solution is reached that cannot be completed without violating some of the problem constraints, one or several earlier assignments have to be undone. This process of undoing earlier assignments is referred to as *backtracking*. It deteriorates the efficiency of the search procedure and increases the time required to come up with a solution. While the worst-case complexity of backtrack search is exponential, several techniques have been proposed in the literature to reduce its average-case complexity [Dechter 88]:

- *Consistency Enforcing Schemes*: These techniques prune the search space from alternatives that cannot participate in a global solution [Mackworth 85]. There is generally a tradeoff between the amount of consistency enforced in each search state[1] and the savings achieved in search time.
- *Look-ahead Schemes*: variable/value ordering heuristics [Bitner 75, Haralick 80, Purdom 83, Dechter 88, Fox 89, Sadeh 91] help judiciously decide which variable to instantiate next and which value to assign to that variable. By first instantiating difficult variables, the system increases its chances of completing the current partial solution without backtracking [Haralick 80, Fox 89, Sadeh 91]. Good value ordering heuristics reduce backtracking by selecting values that are expected to participate in a large number of solutions [Dechter 88, Sadeh 91].
- *Look-back Schemes*: [Stallman 77, Doyle 79, Gaschnig 79, Dechter 89a] While it is possible to design consistency enforcing schemes and look-ahead schemes that are, on the average, very good at efficiently reducing backtracking, it is generally impossible to efficiently guarantee backtrack-free search. Look-back schemes are designed to help the system recover from deadend states and, if possible, learn from past mistakes .

[1]A search state is associated with each partial solution. Each search state defines a new CSP whose variables are the variables that have not yet been instantiated and whose constraints are the initial problem constraints along with constraints reflecting current assignments.

Our earlier work focused on developing efficient consistency enforcing techniques and efficient look-ahead techniques for job shop scheduling CSPs [Sadeh 88, Sadeh 89, Fox 89, Sycara 91, Sadeh 90, Sadeh 91, Sadeh 92]. In this paper, we combine these techniques with new look-back schemes. These schemes are shown to further reduce the average complexity of the search procedure. They also enable our system to efficiently solve problems that could not be efficiently solved otherwise. Finally, experimental results indicate that these techniques are more effective at solving job shop scheduling problems than other look-back schemes advocated in the literature.

The simplest deadend recovery strategy consists in going back to the most recently instantiated variable with at least one alternative value left, and assigning a different value to that variable. This strategy is known as *chronological backtracking*. Often the source of the current deadend is not the most recent assignment but an earlier one. Because it typically modifies assignments that have no impact on the conflict at hand, chronological backtracking often returns to similar deadend states. When this happens, search is said to be *thrashing*. Thrashing can be reduced using backjumping schemes that attempt to backtrack all the way to one of the variables at the source of the conflict [Gaschnig 79]. Search efficiency can be further improved by learning from past mistakes. For instance, a system can record earlier conflicts in the form of new constraints that will prevent it from repeating earlier mistakes [Stallman 77, Doyle 79]. Dependency-directed backtracking is a technique incorporating both backjumping and constraint recording [Stallman 77]. Although dependency-directed backtracking can greatly reduce the number of search states that need to be explored, this scheme is often impractical due to its exponential worst-case complexity (both in time and space). For this reason, simpler techniques have been developed that approximate dependency-directed backtracking. *Graph-based backjumping* reduces the amount of book-keeping required by full-blown backjumping by assuming that any two variables directly connected by a constraint may have been assigned conflicting values [Dechter 89a][2]. *N-th order deep and shallow learning* only record conflicts involving N or fewer variables [Dechter 86].

Graph-based backjumping works best on CSPs with sparse constraint graphs [Dechter 89a]. Instead, job shop scheduling problems have highly interconnected constraint graphs. Furthermore graph-based backjumping does not increase search efficiency when used in combination with forward checking [Haralick 80] mechanisms or stronger consistency enforcing mechanisms such as those entailed by job shop scheduling problems [Sadeh 91]. Experiments reported at the end of this paper also suggest that N-th order deep and shallow learning techniques often fail to improve search efficiency when applied to job shop scheduling problems. This is because these techniques use constraint size as the only criterion to decide whether or not to record earlier failures. When they limit themselves to small-size conflicts, they fail to record some important constraints. When they do not, their complexities become prohibitive.

Instead, this paper presents three look-back techniques that have yielded good results on job shop scheduling problems:

1. *Dynamic Consistency Enforcement* (DCE): a selective dependency-directed scheme that dynamically focuses its effort on critical resource subproblems,

[2]Two variables are said to be "*connected*" by a constraint if they both participate in that constraint.

2. *Learning From Failure* (LFF): an adaptive scheme that suggests new variable orderings based on earlier conflicts,

3. *Heuristic Backjumping* (HB) a scheme that gives up searching areas of the search space that require too much work.

Related work in scheduling includes that of Prosser and Burke who use N-th order shallow learning to solve one-machine scheduling problems [Burke 89], and that of Badie et al. whose system implements a variation of deep learning in which a minimum set is heuristically selected as the culprit [Badie et al 90].

The remainder of this paper is organized as follows. Section 2 provides a more formal definition of the job shop CSP. Section 3 describes the backtrack search procedure considered in this study. Sections 4, 5 and 6 successively describe each of the three backtracking schemes developed for this study. Experimental results are presented in section 7. Section 8 summarizes the contributions of this paper.

Appendix I presents additional experimental results obtained on a testsuite first introduced in [Sadeh 91].

2. The Job Shop Constraint Satisfaction Problem

The job shop scheduling problem requires scheduling a set of jobs $J=\{j_1,...,j_n\}$ on a set of physical resources $RES=\{R_1,...,R_m\}$. Each job j_l consists of a set of operations $O^l=\{O^l_1,...,O^l_{n_l}\}$ to be scheduled according to a process routing that specifies a partial ordering among these operations (e.g. O^l_i BEFORE O^l_j).

In the job shop CSP studied in this paper, each job j_l has a release date rd_l and a due-date dd_l between which all its operations have to be performed. Each operation O^l_i has a fixed duration du^l_i and a variable start time st^l_i. The domain of possible start times of each operation is initially constrained by the release and due dates of the job to which the operation belongs. If necessary, the model allows for additional unary constraints that further restrict the set of admissible start times of each operation, thereby defining one or several time windows within which an operation has to be carried out (e.g. one or several shifts in factory scheduling). In order to be successfully executed, each operation O^l_i requires p^l_i different resources (e.g. a milling machine and a machinist) R^l_{ij} $(1 \le j \le p^l_i)$, for each of which there may be a pool of physical resources from which to choose, $\Omega^l_{ij}=\{r^l_{ij1},...,r^l_{ijq^l_{ij}}\}$, with $r^l_{ijk} \in RES$ $(1 \le k \le q^l_{ij})$ (e.g. several milling machines).

More formally, the problem can be defined as follows:

<u>VARIABLES</u>:

A vector of variables is associated with each operation, O^l_i $(1 \le l \le n,\ 1 \le i \le n_l)$, which includes:

1. the *operation start time*, st^l_i, and

2. each *resource requirement*, R^l_{ij}, $(1 \le j \le p^l_i)$ for which the operation has several alternatives.

CONSTRAINTS:

The non-unary constraints of the problem are of two types:

1. *Precedence constraints* defined by the process routings translate into linear inequalities of the type: $st_i^l + du_i^l \leq st_j^l$ (i.e. O_i^l BEFORE O_j^l);

2. *Capacity constraints* that restrict the use of each resource to only one operation at a time translate into disjunctive constraints of the form: $(\forall p \forall q \, R_{ip}^k \neq R_{jq}^l) \vee st_i^k + du_i^k \leq st_j^l \vee st_j^l + du_j^l \leq st_i^k$. These constraints simply express that, unless they use different resources, two operations O_i^k and O_j^l cannot overlap[3].

Additionally, our model can accommodate unary constraints that restrict the set of possible values of individual variables. These constraints include non-relaxable due dates and release dates, between which all operations in a job need to be performed. In fact, our model can accommodate any type of unary constraint that further restricts the set of possible start times of an operation.

Time is assumed discrete, i.e. operation start times and end times can only take integer values. Each resource requirement R_{ij}^l has to be selected from a set of resource alternatives, $\Omega_{ij}^l \subseteq RES$.

OBJECTIVE:

In the job shop CSP studied in this paper, the objective is to come up with a feasible solution as fast as possible. Notice that this objective is different from simply minimizing the number of search states visited. It also accounts for the time spent by the system deciding which search state to explore next.

3. The Search Procedure

A depth-first backtrack search procedure is considered, in which search is interleaved with the application of consistency enforcing mechanisms and variable/value ordering heuristics that attempt to steer clear of deadend states. Search proceeds according to the following steps:

1. If all operations have been scheduled then stop, else go on to 2;

2. Apply the *consistency enforcing* procedure;

3. If a deadend is detected then *backtrack* (i.e. select an alternative if there is one left and go back to 1, else stop and report that the problem is infeasible), else go on to step 4;

4. Select the next operation to be scheduled (*variable ordering* heuristic);

5. Select a promising reservation for that operation (*value ordering* heuristic);

6. Create a *new search state* by adding the new reservation assignment to the current partial schedule. Go back to 1.

[3]These constraints have to be generalized when dealing with resources of capacity larger than one.

The default consistency enforcing scheme and variable/value ordering heuristics used in the procedure are the ones described in [Sadeh 91]:

Consistency Enforcement: The consistency enforcing procedure is a hybrid procedure that differentiates between precedence constraints and capacity constraints. It guarantees that backtracking only occurs as the result of capacity constraint violations. Essentially, consistency with respect to precedence constraints is enforced by updating in each search state a pair of earliest/latest possible start times for each unscheduled operation. Consistency enforcement with respect to capacity constraints tends to be significantly more expensive due to the disjunctive nature of these constraints. For capacity constraints, a forward checking type of consistency checking is generally carried out by the system. Whenever a resource is allocated to an operation over some time interval, the forward checking procedure checks the set of remaining possible start times of other operations requiring that resource, and removes those start times that would conflict with the new assignment. The system further checks for consistency with respect to a set of redundant capacity constraints, which can be quickly enforced in each search state. This includes checking that no two unscheduled operations totally rely on the same resource over overlapping time intervals[4].

Variable/Value Ordering Heuristics: The default variable/value ordering heuristics used by the search procedure are the *Operation Resource Reliance* (ORR) variable ordering heuristic and *Filtered Survivable Schedules* value ordering heuristic described in [Sadeh 91]. The ORR variable ordering heuristic aims at reducing backtracking by first scheduling difficult operations, namely operations whose resource requirements are expected to conflict with the resource requirements of other operations. The FSS value ordering heuristic is a least constraining value ordering heuristic. It attempts to further reduce backtracking by assigning reservations that are expected to be compatible with a large number of schedules.

These default consistency enforcing schemes and variable/value ordering heuristics have been reported to outperform several other schemes described in the literature, both generic CSP heuristics and specialized heuristics designed for similar scheduling problems [Sadeh 91, Sadeh 92]. They seem to provide a good compromise between the efforts spent enforcing consistency, ordering variables, or ranking assignments for a variable and the actual savings obtained in search time. Nevertheless, the job shop CSP is NP-complete and, hence, these efficient procedures are not sufficient to guarantee backtrack-free search.

The remainder of this paper describes new backtracking schemes that help the system recover from deadend states. It will be seen that, when the default consistency enforcing scheme and/or variable ordering scheme are not sufficient to stay clear of deadends, look-back mechanisms can be devised that will modify these schemes so as to avoid repeating past mistakes (i.e.so as to avoid reaching similar deadend states).

[4]See [Sadeh 91] for further details.

4. Dynamic Consistency Enforcement (DCE)

Backtracking is generally an indication that the default consistency enforcing scheme and/or variable/value ordering heuristics used by the search procedure are insufficient to deal with the subproblems at hand. Consequently, if search keeps on relying on the same default mechanisms after reaching a deadend state, it is likely to start thrashing. Experiments reported in [Sadeh 91, Sadeh 92], in which search always used the same set of consistency enforcing procedures and variable/value ordering heuristics, clearly illustrated this phenomenon. Search in these experiments exhibited a dual behavior. The vast majority of the problems fell in either of two categories: a category of problems that were solved with no backtracking whatsoever (by far the largest category) and a category of problems that caused the search procedure to thrash.

Theoretically, thrashing could be eliminated by enforcing full consistency in each search state. Clearly, such an approach is impractical as it would amount to performing a complete search. Instead, our approach consists in (1) heuristically identifying one or a few small subproblems that are likely to be at the source of the conflict, (2) determining how far to backtrack by enforcing full consistency among the variables in these small subproblems, and (3) recording conflict information for possible reuse in future backtracking episodes. This approach is operationalized in the context of a backtracking scheme called *Dynamic Consistency Enforcement* (DCE). Given a deadend state and a history of earlier backtracking episodes within the same search space (i.e. while working on the same problem), this technique dynamically identifies small critical resource subproblems expected to be at the source of the current deadend. DCE then backtracks, undoing assignments in a chronological order, until a search state is reached, within which consistency has been fully restored in each critical resource subproblem (i.e. consistency with respect to capacity constraints in these subproblems). Experimental results reported in Section 7 suggest that often, by *selectively* checking for consistency in small resource subproblems, DCE can quickly recover from deadends. The remainder of this section further describes the mechanics of this heuristic.

4.1. Identifying Critical Resource Subproblems

The critical resource subproblems used by DCE consist of groups of operations participating in the current conflict along with groups of critical operations identified during earlier backtracking episodes involving the same resources. Below, we refer to the group of (unscheduled) operations identified by the default consistency enforcing mechanism as having no possible reservations left as the *Partial Conflicting Set* of operations (PCS). In order to restore consistency, the search procedure needs to at least go back to a search state in which each PCS operation has one or more possible reservations. Often, however, this is not sufficient, as other operations contribute to the conflict. DCE attempts to identify these other operations by maintaining a data structure of critical resource subproblems identified during earlier backtracking episodes. Below, we refer to this data structure as the *Former Dangerous Groups* of operations (FDG). Details on how this data structure is created and maintained are provided in Subsection 4.3.

For each capacity constraint violation among operations in the PCS, DCE checks the FDG data structure and retrieves all related resource subproblems. A resource subproblem in the FDG is considered related to a capacity constraint violation in the PCS if, in an earlier backtracking episode, operations in that resource subproblem were involved in a capacity constraint violation on the same resource and over a "close" time interval. A system parameter is used to determine if two resource conflicts are "close". In the experiments reported at the end of this paper, two conflicts were considered close if the distance separating them was not greater than twice the average operation duration. Related critical subproblems identified by inspecting the FDG data structure are then merged with corresponding operations in the PCS to form a new set of one or more critical resource subproblems. Below we refer to the overall set of operations identified by inspecting the FDG as the *Dangerous Group* of operations (DG) for the conflict at hand. Finally, at each level, while backtracking, the set consisting of the union of the PCS, the DG and the set of undone operations up to that level is referred to as the *Deadend Operation Set* (DOS). Like the FDG, the DOS is organized in groups of resource subproblems consisting of operations contending for the same resource over close or overlapping time intervals.

4.2. Backtracking While Selectively Enforcing Consistency

Once an initial DOS has been identified, DCE backtracks, undoing assignments in a chronological order, until it reaches a search state in which consistency is restored within each of the resource subproblems defined by operations in the DOS. While restoring consistency within each of these resource subproblems is a necessary condition to backtrack to a consistent search state, it is clearly not a sufficient one. In other words, the effectiveness of DCE critically depends on its ability to heuristically focus on the right resource subproblems[5].

During backtracking, unscheduled operations are merged into corresponding resource subproblems in the DOS. Because full consistency checking can be expensive on large subproblems, if a resource subproblem in the DOS becomes too large, k-consistency is enforced instead of full-consistency, where k is a parameter of the system [Freuder 82]. In the experiments reported at the end of this paper, k was set to 4. At the end of a backtracking episode, DOS has maximum size, call it DOS_{max}. Assuming that the procedure was able to backtrack to a consistent search state[6], DOS_{max} contains all the operations at the origin of the deadend (and often more). DOS_{max} is then saved for later use in the FDG data structure. Additional details regarding the management of this data structure are provided in the next subsection. If a related backtracking episode is later encountered by the system, DOS_{max} can then be retrieved and serve as the DG for this new episode.

The behavior of the DCE procedure is illustrated in Figure 1. Each node represents a search state, labeled by the operation that was last scheduled to reach that state, the resource allocated to that operation, and the operation's start time. In this example, search is assumed to have reached a deadend at depth D_5. Operations in the PCS are those operations whose domains of possible start times were identified as empty at depth D_5 due to capacity constraint violations. Upon encountering a deadend at D_5, DCE backtracks to D_4 and performs full consistency checking with respect to capacity constraints on the set of operations $DOS_4=PCS \cup DG \cup \{O_m\}$. If there are still capacity constraint violations at D_4, operation O_l is undone, and full consistency checking is performed on the new DOS, namely $DOS_3=PCS \cup DG \cup \{O_l, O_m\}$. The procedure is

[5]Notice that DCE is not expected to be very effective at recovering from complex conflicts involving interactions between multiple resource subproblems. Instead, in Section 6, we present a Backjumping Heuristic that appears more appropriate for dealing with these complex conflicts.

[6]Clearly, while this is not guaranteed, experimental results suggest that this is often the case.

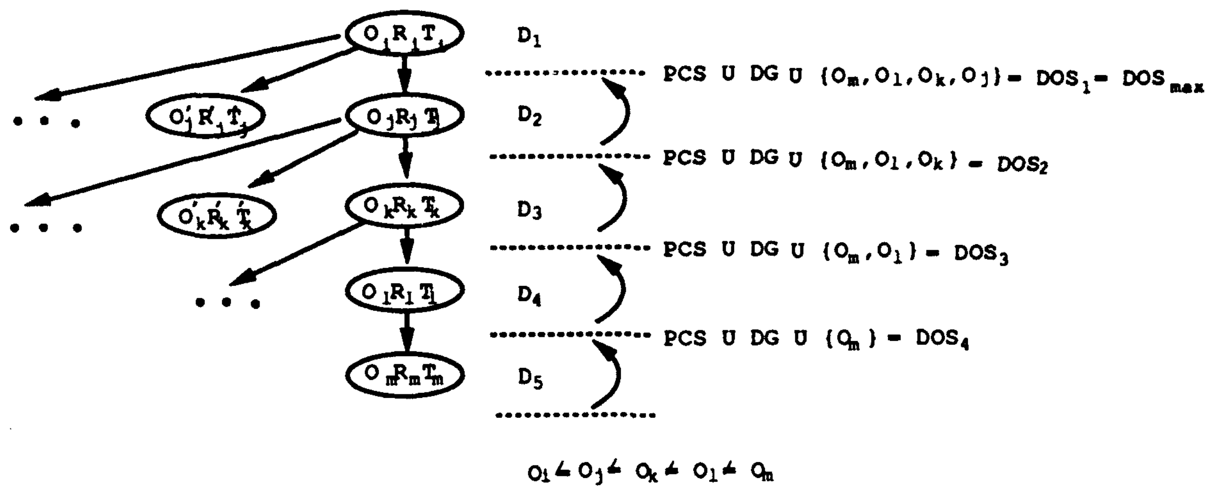
D1
D2
D3
D4
D5
PCS U DG U {Om, O1, Ok, Oj} = DOS1 = DOSmax
PCS U DG U {Om, O1, Ok} = DOS2
PCS U DG U {Om, O1} = DOS3
PCS U DG U {Om} = DOS4
O1 ≤ Oj ≤ Ok ≤ O1 ≤ Om

Figure 1: The DCE Backtracking Scheme.

repeated until a consistent DOS is found ($DOS_{max}=DOS_1$ in this example).

4.3. Storing Information About Past Backtracking Episodes

The purpose of the *Former Dangerous Groups* of operations (FDG) maintained by the system is to help determine more efficiently and more precisely the scope of each deadend by focusing on critical resource subproblems. Each group of operations in the FDG consists of operations that are in high contention for the allocation of a same resource. Accordingly, whenever, a conflict is detected that involves some of the operations in one group, the backtracking procedure checks for consistency among *all* operations in that group.

The groups of operations in the FDG are built from the *Deadend Operation Sets* (DOS) obtained at the end of previous backtracking episodes ($\mathrm{DOS}_{\mathrm{max}}$). Indeed, whenever a backtracking episode is completed, $\mathrm{DOS}_{\mathrm{max}}$ is expected to contain all the conflicting operations at the origin of this episode. Generally, $\mathrm{DOS}_{\mathrm{max}}$ may involve one or several resource subproblems (i.e. groups of operations requiring the same resource). Each one of these subproblems is merged with *related* subproblems currently stored in the FDG. If there is no related group in FDG, the new group is separately added to the data structure. Finally, as operations are scheduled, they are removed from the FDG.

4.4. Additional "Watch Dog" Consistency Checks

Because groups of operations in the FDG are likely deadend candidates, our system further performs simple "watch dog" checks on these dynamic groups of operations.

More specifically, for each group G of operations in FDG, the system performs a rough check to see if the resource can still accommodate all the operations in the group. This is done using redundant constraints of the form:

$$Max(lst_i^l + du_i^l, O_i^l \in G) - Min(est_i^l, O_i^l \in G) \geq \sum_{O_i^l \in G} du_i^l$$

where est_i^l and lst_i^l are respectively the earliest and latest possible start times of O_i^l in the current search state.

Whenever such a constraint is violated, an inconsistency has been detected. Though very simple and inexpensive, these checks enable to catch inconsistencies involving large groups of operations that would not be immediately detected by the default consistency mechanisms some inconsistencies can still escape these rough checks.

5. Learning From Failures (LFF)

Encounter of a deadend is also often an indication that the default variable ordering was not adequate for dealing with the subproblem at hand. Typically the operations participating in the deadend turn out to be more difficult than the operations selected by the default variable ordering heuristic. It is therefore a good idea to first schedule the operations participating in the conflict that was just resolved. *Learning From Failure* (LFF) is an adaptive procedure that overrides the default variable ordering in the presence of conflicts.

After recovering from a deadend (i.e. after backtracking all the way to an apparently consistent search state), LFF uses the Partial Conflicting Set (PCS) of the deadend to reorganize the order in which operations will be rescheduled and make sure that operations in the PCS are scheduled first. This is done using a quasi-stack, QS, on which operations in the PCS are pushed in descending order of domain size (operations with more available start times go first). If a candidate operation is already in QS, i.e. it is encountered for a second time, it is pushed again as though it had a smaller domain. This orders operations in terms of their criticality (most critical operation on top) so as to ensure that, as QS is popped, the most critical operations will be scheduled first. When QS becomes empty, the search procedure switches back to its default variable ordering heuristic.

6. A Backjumping Heuristic

Traditional backtrack search procedures only undo decisions that have been proven to be inconsistent. Proving that an assignment is inconsistent with others can be very expensive, especially when dealing with large conflicts. Graph-based backjumping and N-th order shallow/deep learning attempt to reduce the complexity of full-blown dependency-directed backtracking by either simplifying the process of identifying inconsistent decisions (e.g. based on the topology of the constraint graph) or restricting the size of the conflicts that can be detected. The Dynamic Consistency Enforcement (DCE) procedure described in Section 6 also aims at reducing the complexity of identifying the source of a conflict by dynamically focusing its effort on small critical subproblems. Because these techniques focus on smaller conflicts, they all have problems dealing with more complex conflicts involving a large number of variables[7]. It may in fact turn out that the only effective way to deal with more complex conflicts is by using heuristics that undo decisions not because they have been proved to be inconsistent but simply because they appear overly restrictive. This is the approach taken in the backjumping heuristic described in this section. Clearly, the resulting search procedure is no longer complete and may fail to find solutions to feasible problems.

[7]Clearly, there are some conflicts involving large numbers of variables that are easy to catch, as illustrated by the watch dog checks described in Section 4.

Texture measures such as the ones described in [Fox 89] could be used to estimate the tightness of different search states, for instance, by estimating the number of global solutions compatible with each search state[8]. Assignments leading to much tighter search states would be prime candidates to be undone when a complex conflict is suspected. The *Backjumping Heuristic* (BH) used in this study is simpler and, yet, often seems to get the job done. Whenever the system starts thrashing, this heuristic backjumps all the way to the first search state and simply tries the next best value (i.e. reservation) for the critical operation in that state (i.e. the first operation selected by the variable ordering heuristic). BH considers that the search procedure is thrashing, and hence that it is facing a complex conflict, when more than θ assignments had to be undone since the last time the system was thrashing or since the procedure began, if no thrashing occurred earlier. θ is a parameter of the search procedure.

7. Experimental Results

Two sets of 40 scheduling problems each were generated that differed in the number of major bottlenecks (one and two major bottlenecks respectively). Each problem had 50 operations and 5 resources (i.e., 10 jobs). All jobs were released at the same time and had to be completed by the same due date. In each problem, the common due date was set so that all operations had to be scheduled within a rather tight estimate of the problem makespan (see [Sadeh 91] for details). These are the conditions in which the default variable/value ordering and consistency enforcing schemes work least effectively (see study reported in [Sadeh 91]). Among these 80 problems, we only report performance on problems in which the default schemes were not sufficient to guarantee backtrack-free search[9]. This leaves 16 scheduling problems with one bottleneck, and 15 with two bottlenecks. Additional results are also presented in Appendix I. We successively report the results of two studies. The first study compares the performance of three complete backtrack schemes: chronological backtracking, 2nd-order deep learning, and the procedure combining the DCE and LFF backtrack schemes described in Section 4 and 5. The second study compares the complete search procedure using the DCE and LFF backtracking schemes with an incomplete search procedure combining DCE and LFF with the Backjumping Heuristic (BH) described in Section 6.

7.1. Comparison of Complete Search Procedures

The two "intelligent" backtracking techniques, DCE and LFF are complementary and were used in combination, denoted by DCE & LFF[10]. Each of the problems in the experiment set was run using chronological backtracking, 2nd-order deep learning [Dechter 89b] and the DCE & LFF procedures advocated in Section 4 and 5. The results reported here were obtained using a search limit of 500 nodes and a time limit of 1800 seconds (except for deep learning, for which

[8]A search state whose partial solution is compatible with a large number of global solutions is a loosely constrained search state, whereas one compatible with a small number of global solutions is tightly constrained.

[9]Clearly, performance on problems that do not require backtracking is of no interest to this study. Our backtracking schemes never get invoked on these problems, and hence CPU time remains unchanged.

[10]Besides the experiments reported below, additional experiments were performed to assess the benefits of using DCE and LFF separately. These experiments show that both techniques contribute to the improvements reported in this section.

Table 1: Results of One-Bottleneck Experiments.

Exp. No.	Chronological Backtracking			DCE & LFF			Deep Learning		
	No. of Nodes	CPU (sec)	Result	No. of Nodes	CPU (sec)	Result	No. of Nodes	CPU (sec	Result
1	500	1427	F	122	1232	S*	500	5756	F
2	500	1587	F	500	1272	F	500	5834	F
3	74	148	S	63	117	S	25	36000	F
4	69	152	S	52	120	S	69	391	S
5	500	1407	F	65	134	S	500	11762	F
6	500	1469	F	500	1486	F	500	8789	F
7	500	1555	F	59	130	S	500	9681	F
8	500	1705	F	41	145	S*	500	9560	F
9	53	108	S	53	102	S	53	122	S
10	500	1529	F	500	1536	F	500	9114	F
11	500	1460	F	85	1800	F	500	14611	F
12	500	1694	F	500	1131	F	500	21283	F
13	51	109	S	51	81	S	51	88	S
14	500	1762	F	63	138	S	500	18934	F
15	500	1798	F	69	142	S	500	9600	F
16	500	1584	F	500	1183	F	65	36000	F

S: Solved ; F: Failure; S*: Proved infeasible
Time Limit: 1800 sec (Except Deep Learning)
Node Limit: 500

the time limit was increased to 36,000 seconds[11]). All CPU times reported below were obtained on a DECstation 5000 running Knowledge Craft on top of Allegro Common Lisp. Comparison between C and Knowledge Craft implementations of similar variable and value ordering heuristics suggests that the code would run about 30 times faster in C [Sadeh 93].

Results for the one-bottleneck problems are reported in Table 1. Chronological backtracking solved only 4 problems out of 16. Interestingly enough, deep learning showed no improvement over chronological backtracking either in the number of problems solved or in CPU time. As a matter of fact, deep learning was even too slow to find solutions to some of the problems solved by chronological backtracking. This is attributed to the fact that the constraints in job shop scheduling are more tightly interacting than those in the zebra problem, where the improvement of deep learning over naive backtracking was originally ascertained. On the other hand, DCE & LFF solved 10 problems out of 16 (2 out of these 10 problems were successfully proven infeasible). As expected, by focusing on a small number of critical subproblems, DCE & LFF is able to discover larger more useful conflicts than 2nd-order deep learning, while requiring only a fraction of the time. Another observation is that DCE & LFF expanded fewer search states than chronological backtracking for the problems that chronological backtracking solved. However, each of the DCE & LFF expansions took slightly more CPU time, due to the higher level of consistency enforcement.

Results for the set of two-bottleneck problems are reported in Table 2. Similar results are observed here again: deep learning shows no improvement over chronological backtracking and seems significantly slower. The difference between chronological backtracking and DCE&LFF is not as impressive as in the first set of experiments. This is probably because both bottlenecks may have capacity conflicts at the same time. DCE & LFF may then have problems determining which one to consider first. As can be seen in Table 2, chronological backtracking solved 7 out of 15 problems, whereas DCE & LFF solved 8 out of 15. On the problems solved by both chronological backtracking and DCE & LFF, DCE & LFF turned out to be slightly faster overall.

[11]This was motivated by the fact that our implementation of deep learning may not be optimal.

Table 2: Results of Two-bottleneck Experiments

Exp. No.	Chronological Backtracking			DCE & LFF			Deep Learning		
	No. of Nodes	CPU (sec)	Result	No. of Nodes	CPU (sec)	Result	No. of Nodes	CPU (sec)	Result
1	500	1139	F	113	1800	F	18	36000	F
2	500	1444	F	425	1800	F	115	36000	F
3	84	175	S	109	202	S	84	811	S
4	56	123	S	56	112	S	56	213	S
5	51	101	S	51	113	S	13	36000	F
6	500	1531	F	321	1800	F	328	36000	F
7	500	1775	F	500	1357	F	500	2793	F
8	52	102	S	52	115	S	33	36000	F
9	500	1634	F	247	974	S	500	1519	F
10	500	1676	F	91	1800	F	26	36000	F
11	66	163	S	59	104	S	66	2240	S
12	56	139	S	58	104	S	58	281	S
13	54	129	S	52	91	S	54	28900	S
14	500	1676	F	346	1800	F	500	9031	F
15	500	1522	F	324	1800	F	296	36000	F

S: Solved ; F: Failure; S*: Proved infeasible
Time Limit : 1800 sec. (36000 sec. for Deep Learning)
Node Limit : 500

7.2. Complete vs. Incomplete Search Procedures

Table 3: Results of One-bottleneck Experiments.

Exp. No.	DCE & LFF			DCE & LFF & BH		
	No. of Nodes	CPU (sec)	Result	No. of Nodes	CPU (sec)	Result
1	122	1232	S*	350	1800	F
2	500	1272	F	203	1124	S
3	63	117	S	63	123	S
4	52	120	S	52	116	S
5	65	134	S	65	144	S
6	500	1486	F	127	424	S
7	59	130	S	59	125	S
8	41	145	S*	457	1800	F
9	53	108	S	53	100	S
10	500	1536	F	67	170	S
11	85	1800	F	74	170	S
12	500	1131	F	164	616	S
13	51	81	S	51	92	S
14	63	138	S	63	149	S
15	69	142	S	69	158	S
16	500	1183	F	156	524	S

S: Solved ; F: Failure; S*: Proved infeasible
Time Limit: 1800 sec. Node Limit: 500

Table 3 and 4 compare the performance of the complete search procedure based on DCE & LFF against that of an incomplete search procedure using DCE & LFF in combination with the Backjumping Heuristic (BH) described in Section 6. While DCE & LFF was able to solve only 10 out of 16 one-bottleneck problems and 8 out 15 two-bottleneck problems, DCE & LFF combined with BH solved 14 one-bottleneck problems and 13 two-bottleneck problems. The only one-bottleneck problems that were not solved by DCE & LFF & BH are the two infeasible problems identified by the complete search procedure DCE & LFF. This is hardly a surprise. While the addition of BH to DCE & LFF enables the search procedure to solve a larger number of problems, it also makes the procedure incomplete (i.e. infeasible problems can no longer be identified). Additional experiments combining BH with a simple chronological backtracking scheme produced results that were not as good as those obtained by DCE & LFF & BH. This indicates that both BH and DCE & LFF contribute to the performance increases observed in

Table 4: Results of Two-bottleneck Experiments

Exp. No.	DCE & LFF			DCE & LFF & BH		
	No. of Nodes	CPU (sec)	Result	No. of Nodes	CPU (sec)	Result
1	113	1800	F	151	456	S
2	425	1800	F	371	1780	S
3	109	202	S	95	210	S
4	56	112	S	56	108	S
5	51	113	S	51	97	S
6	321	1800	F	420	1800	F
7	500	1357	F	159	534	S
8	52	115	S	52	96	S
9	247	974	S	423	1705	S
10	91	1800	F	440	1800	F
11	59	104	S	59	113	S
12	58	104	S	58	112	S
13	52	91	S	52	102	S
14	346	1800	F	239	512	S
15	324	1800	F	73	195	S

S: Solved ; F: Failure; S*: Proved infeasible
Time Limit: 1800 sec. Node Limit: 500

Table 3 and 4.

Results on two-bottleneck problems (See Table 4) also suggest that the impact of the backjumping heuristic is particularly effective on these problems. This is attributed to the fact that two-bottleneck problems give rise to more complex conflicts. Identifying the assignments participating in these more complex conflicts may simply be too difficult for any exact backtracking scheme. Instead, because it can undo assignments that are not provably wrong but simply appear overly restrictive, BH seems more effective at dealing with these more complex conflicts.

8. Concluding Remarks

We have presented three "intelligent" backtracking schemes for the job shop scheduling CSP:

1. *Dynamic Consistency Enforcement* (DCE), a dependency-directed scheme, that dynamically focuses its effort on small critical subproblems,
2. *Learning From Failure* (LFF), which modifies the order in which variables are instantiated based on earlier conflicts, and
3. a *Backjumping Heuristic* which, when thrashing occurs, can undo assignments that are not provably inconsistent but appear overly restrictive.

The significance of this research is twofold:

1. Job shop scheduling problems with non-relaxable time windows have multiple applications, including both manufacturing and space-related applications. We have shown that our schemes combined with powerful techniques that we had previously developed (1) further reduce the average complexity of backtrack search, and (2) enable our system to efficiently solve problems that could not be solved otherwise due to excessive computational cost. While the results reported in this study were obtained on problems that require finding a feasible schedule, the backtracking schemes presented in this paper can also be used on optimization versions of the scheduling problem, such as those discussed in [Sadeh 93].
2. This research also points to the shortcomings of dependency-directed backtracking

schemes advocated earlier in the literature. In particular, comparison with N-th order deep learning indicates that this technique failed to improve performance on our set of job shop scheduling problems. This is because N-th order deep learning uses constraint size as the only criterion to decide whether or not to record earlier failures. When deep learning limits itself to small-size conflicts, it fails to record some important constraints; when it considers conflicts of larger size, its computational complexity becomes prohibitive. Traditional backtracking schemes never undo assignments unless they can prove that they are at the source of the conflict. When dealing with large complex conflicts, proving that a particular assignment should be undone can be very expensive. Instead, our experiments suggest that, when thrashing cannot easily be avoided, it is often a better idea to use backjumping heuristics that undo decisions simply because they *appear* overly restrictive. When using such heuristics, search completeness can no longer be guaranteed.

Appendix I

For reference, this appendix reports additional experimental results obtained on a testsuite of 60 job shop scheduling problems first introduced in [Sadeh 91]. The testsuite consists of 6 groups of 10 problems. Each problem requires scheduling 10 jobs on 5 resources (50 operations total). Each job has a linear process routing specifying a sequence in which it has to visit each one of the five resources. This sequence varies from one job to another, except for a predetermined number of bottleneck resources (one or two in these experiments) which are always visited after the same number of steps. The six groups of problems were obtained by varying two parameters:

1. the number of apriori bottlenecks (BTNK): one (BTNK=1) or two (BTNK=2), and
2. the spread (SP) of release and due dates between which each job has to be scheduled: wide (SP=W), narrow (SP=N), or null (SP = 0).

Additional details on how these scheduling problems were obtained can be found in [Sadeh 91].

Table I-1 compares the performance of chronological backtracking and DCE & LFF & BH on the 60 problems. For each problem, search was stopped if it required more than 500 search states. Performance in each problem category is reported along three dimensions:

1. *Search efficiency*: the average ratio of the number of operations to be scheduled over the total number of search states that were explored. In the absence of backtracking, only one search state is generated for each operation, and hence search efficiency is equal to 1.
2. *Number of experiments* solved in less than 500 search states each.
3. *CPU seconds*: this is the average CPU time required to solve a problem. When a solution could not be found, this time was approximated as the CPU time taken to explore 500 search states (this approximation was only used for Chronological Backtracking, since DCE&LFF&BH solved all problems). All CPU times were obtained on a DECstation 5000 running Knowledge Craft on top of Allegro Common Lisp. Experimentation with a variation of the system written in C++ suggests that the search procedure would run about 30 times faster if reimplemented in this language [Sadeh 93].

The results indicate that DCE&LFF&BH consistently outperformed the chronological backtracking scheme in terms of CPU time, search efficiency and number of problems solved. The most impressive performance improvements were obtained on the most difficult problems (SP=N and SP=0). In particular, on problems with SP=0 and BK=1, DCE&LFF&BH solved 40% more problems than the chronological backtracking scheme and, on the average, proved to be 3.5 times faster. Overall, while chronological backtracking failed to solve 8 problems out of 60, DCE&LFF&BH was able to efficiently solve all 60 problems, and, on the average, was almost twice as fast as the chronological backtracking scheme. These results further confirm the effectiveness of the backtracking scheme described in this paper.

		Chronological	DCE&LFF&BH
SP=W BTNK=1	Search Efficiency	0.96	0.96
	Nb. exp. solved (out of 10)	10	10
	CPU seconds	88.5	90.5
SP=W BTNK=2	Search Efficiency	0.99	0.99
	Nb. exp. solved (out of 10)	10	10
	CPU seconds	93	95
SP=N BTNK=1	Search Efficiency	0.78	0.91
	Nb. exp. solved (out of 10)	8	10
	CPU seconds	331.5	106
SP=N BTNK=2	Search Efficiency	0.87	0.93
	Nb. exp. solved (out of 10)	9	10
	CPU seconds	184	119.5
SP=0 BTNK=1	Search Efficiency	0.73	0.88
	Nb. exp. solved (out of 10)	7	10
	CPU seconds	475	134.5
SP=0 BTNK=2	Search Efficiency	0.82	0.84
	Nb. exp. solved (out of 10)	8	10
	CPU seconds	300.5	226.5
Overall Performance	Search Efficiency	0.86	0.92
	Nb. exp. solved (out of 60)	52	60
	CPU seconds	245.5	128.7

Table I-1: Comparison of Chronological Backtracking and DCE&LFF&BH on 6 sets of 10 job shop problems.

References

[Badie et al 90] C. Badie and G. Bel and E. Bensana and G. Verfaillie.
Operations Research and Artificial Intelligence Cooperation to solve Scheduling Problems.
In *First International Conference on Expert Planning Systems*. 1990.

[Bitner 75] J.R. Bitner and E.M. Reingold.
Backtrack Programming Techniques.
Communications of the ACM 18(11):651-655, 1975.

[Burke 89] Peter Burke and Patrick Prosser.
A Distributed Asynchronous System for Predictive and Reactive Scheduling.
Technical Report AISL-42, Department of Computer Science, University of Strathclyde, 26 Richmond Street, Glasgow, GI IXH, United Kingdom, October, 1989.

[Dechter 86] Rina Dechter.
Learning while Searching in Constraint Satisfaction Problems.
In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 178-183. 1986.

[Dechter 88] Rina Dechter and Judea Pearl.
Network-Based Heuristics for Constraint Satisfaction Problems.
Artificial Intelligence 34(1):1-38, 1988.

[Dechter 89a] Rina Dechter.
Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition.
Artificial Intelligence 41:273-312, 1989.

[Dechter 89b] Rina Dechter and Itay Meiri and Judea Pearl.
Temporal Constraint Networks.
In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*. 1989.

[Doyle 79] John Doyle.
A Truth Maintenance System.
Artificial Intelligence 12(3):231-272, 1979.

[Fox 89] Mark S. Fox and Norman Sadeh and Can Baykan.
Constrained Heuristic Search.
In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 309-315. 1989.

[Freuder 82] E.C. Freuder.
A Sufficient Condition for Backtrack-free Search.
Journal of the ACM 29(1):24-32, 1982.

[Garey 79] M.R. Garey and D.S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
Freeman and Co., 1979.

[Gaschnig 79] John Gaschnig.
Performance Measurement and Analysis of Certain Search Algorithms.
Technical Report CMU-CS-79-124, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, 1979.

[Golomb 65] Solomon W. Golomb and Leonard D. Baumert.
Backtrack Programming.
Journal of the Association for Computing Machinery 12(4):516-524, 1965.

[Haralick 80] Robert M. Haralick and Gordon L. Elliott.
Increasing Tree Search Efficiency for Constraint Satisfaction Problems.
Artificial Intelligence 14(3):263-313, 1980.

[Mackworth 85] A.K. Mackworth and E.C. Freuder.
The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems.
Artificial Intelligence 25(1):65-74, 1985.

[Purdom 83] Paul W. Purdom, Jr.
Search Rearrangement Backtracking and Polynomial Average Time.
Artificial Intelligence 21:117-133, 1983.

[Sadeh 88] N. Sadeh and M.S. Fox.
Preference Propagation in Temporal/Capacity Constraint Graphs.
Technical Report CMU-CS-88-193, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, 1988.
Also appears as Robotics Institute technical report CMU-RI-TR-89-2.

[Sadeh 89] N. Sadeh and M.S. Fox.
Focus of Attention in an Activity-based Scheduler.
In *Proceedings of the NASA Conference on Space Telerobotics.* January, 1989.

[Sadeh 90] Norman Sadeh, and Mark S. Fox.
Variable and Value Ordering Heuristics for Activity-based Job-shop Scheduling.
In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management, Hilton Head Island, S.C.*, pages 134-144. 1990.

[Sadeh 91] Norman Sadeh.
Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling.
PhD thesis, School of Computer Science, Carnegie Mellon University, March, 1991.

[Sadeh 92] N. Sadeh and M.S. Fox.
Variable and Value Ordering Heuristics for Hard Constraint Satisfaction Problems: an Application to Job Shop Scheduling.
Technical Report CMU-RI-TR-91-23, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213, 1992.

[Sadeh 93]	Norman Sadeh. Micro-Opportunistic Scheduling: The MICRO-BOSS Factory Scheduler. *Intelligent Scheduling*. In Mark Fox and Monte Zweben, Morgan Kaufmann Publishers, 1993.
[Stallman 77]	R. Stallman and G. Sussman. Forward Reasoning and Dependency-directed Backtracking in a Sysem for Computer-aided Circuit Analysis. *Artificial Intelligence* 9:135-196, 1977.
[Sycara 91]	Sycara, K. and Roth, S. and Sadeh, N. and Fox, M. Distributed Constrained Heuristic Search. *IEEE Transactions on System, Man and Cybernetics* 21(6), 1991.
[Walker 60]	R.J. Walker. An Enumerative Technique for a Class of Combinatorial Problems. *Combinatorial Analysis, Proc. Sympos. Appl. Math.* In R. Bellman and M. Hall, American Mathematical Society, Rhode Island, 1960, pages 91-94, Chapter 7.